LINK ANALYSIS

In order to perform link analysis, a real-world network dataset is taken and analysed.

1. Number of Nodes
2. Number of Edges

Graph represented as edge list or adjacency list

Creating adjacency list

```
adjacency_list = defaultdict(list)                    # adjacency list for the graph
nodes = set()                                         # set of nodes
edges = 0                                             # number of edges

# 1st 4 lines in the given text file consists of meta-data.
# Thus, started reading after that
for line in read_file[4:]:
  edges += 1
  a = line.split('\t')
  adjacency_list[a[0]].append(a[1].strip('\n'))
  nodes.add(a[0])
  nodes.add(a[1].strip('\n'))
```

```
[36]  print(len(nodes))
```

```
7115
```

```
[37]  print(edges)
```

```
103689
```

The number of nodes and edges calculated in the code matches with that given in the meta-data.

Creating adjacency matrix

```
[10]  adjacency_matrix = np.zeros([len(nodes),len(nodes)])

      for node_from in adjacency_list:
        i = mapping[node_from]

        for node_to in adjacency_list[node_from]:
          j = mapping[node_to]
          adjacency_matrix[i][j] = 1
```

3. Avg In-degree

Avg. in-degree

> in-degree of a node is equal to the number of non-zero entries in adjacency matrix in the column of that node

```
[13]  1 in_degree_sum = 0
      2
      3 for i in range(0,len(nodes)):
      4   # extracting the column corresponding to node i from the adjacency matrix
      5   l = adjacency_matrix[:,i]
      6   in_degree_sum += np.count_nonzero(l)
      7
      8 avg_in_degree = in_degree_sum/len(nodes)
```

```
[14]  1 print(avg_in_degree)
```

```
14.573295853829936
```

In-degree of a node is equal to the number of non-zero entries in the adjacency matrix in the *column* of that node.

4. Avg. Out-Degree

Avg. out-degree

> out-degree of a node is equal to the number of non-zero entries in adjacency matrix in the row of that node

```
[11]  1 out_degree_sum = 0
      2
      3 for i in range(0,len(nodes)):
      4   # extracting the row corresponding to node i from the adjacency matrix
      5   l = adjacency_matrix[i,:]
      6   out_degree_sum += np.count_nonzero(l)
      7
      8 avg_out_degree = out_degree_sum/len(nodes)
```

```
[12]  1 print(avg_out_degree)
```

```
14.573295853829936
```

Out-degree of a node is equal to the number of non-zero entries in the adjacency matrix in the *row* of that node.

Since it was a directed graph, average in-degree and average out-degree has to be the same. This property is held even in the graph of our dataset. Both average in-degree and out-degree is 14.573295.

5. Node with Max In-degree

Max in-degree

```
[17]  1 in_degree = []
      2
      3 for i in range(0,len(nodes)):
      4   l = adjacency_matrix[:,i]                    # extracting the column corresponding to ith node
      5   in_degree.append((np.count_nonzero(l),i))
      6
      7 max_in = max(in_degree, key = lambda x : x[0])
```

```
[18]  1 for key,value in mapping.items():
      2   if(value == max_in[1]):
      3     print("The node with maximum out-degree is",key)
      4
      5 print("Its out-degree is",max_in[0])
```

```
The node with maximum out-degree is 4037
Its out-degree is 457
```

The node with maximum in-degree is 4037 and has an in-degree of 457.

6. Node with Max out-degree

Max out-degree

```
[15]  1 out_degree = []
      2
      3 for i in range(0,len(nodes)):
      4   l = adjacency_matrix[i,:]              # extracting the row corresponding to ith node
      5   out_degree.append((np.count_nonzero(l),i))
      6
      7 max_out = max(out_degree, key = lambda x : x[0])
```

```
[16]  1 for key,value in mapping.items():
      2   if(value == max_out[1]):
      3     print("The node with maximum out-degree is",key)
      4
      5 print("Its out-degree is",max_out[0])
```

```
The node with maximum out-degree is 2565
Its out-degree is 893
```

The node with maximum out-degree is 2565 and has an out-degree of 893.

7. Density of the network

In order to calculate density of the network, following formula was used:

density = edges / nodes(nodes-1)

The formula was taken from networkx website.

Source:https://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.classes.function.density.html

Network Density

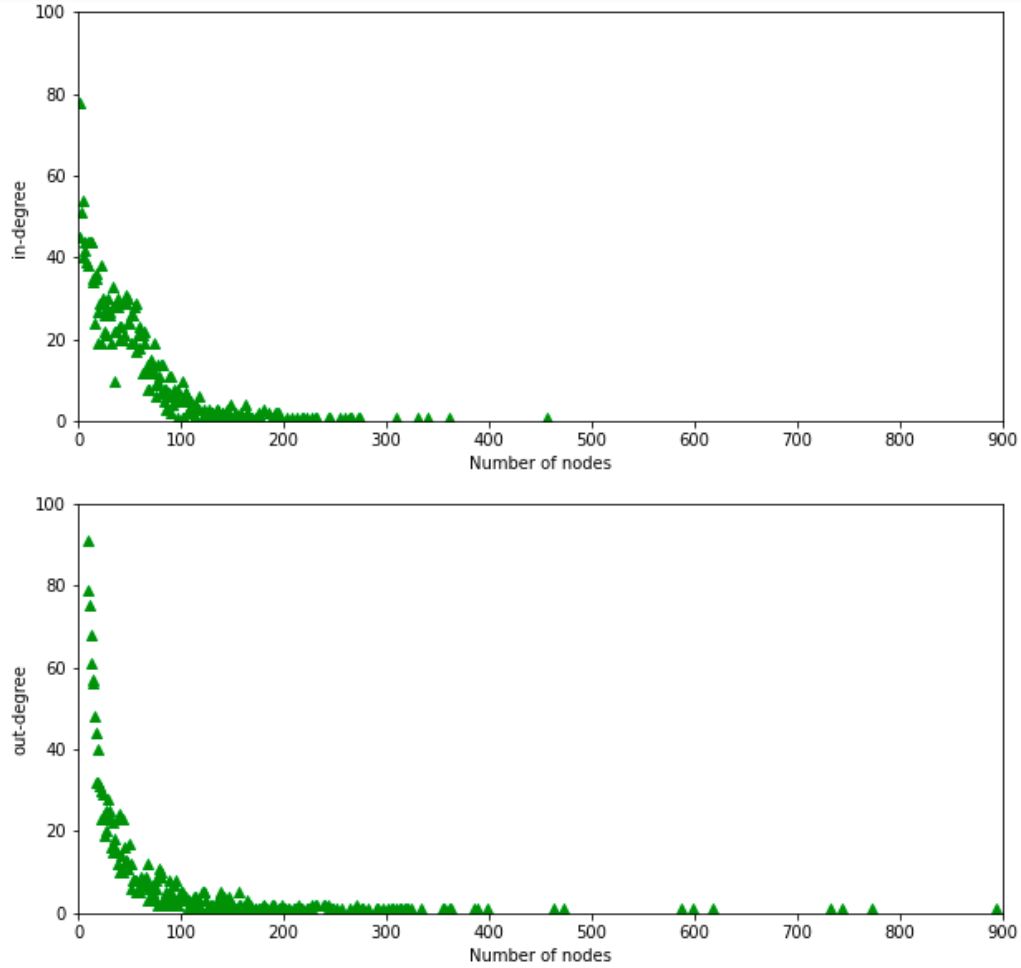
> density = edges / nodes(nodes-1)

```
[19]  1 N = len(nodes)
      2 density = edges/(N * (N-1))
      3 print(density)
```

```
0.0020485375110809584
```

8. Plot degree distribution of the network

[22]



In the above graph, x-axis shows the frequency of the nodes having certain in-degree or out-degree. The in-degree and out-degree are plotted on the y-axis.
We can observe that the number of nodes with ‘0’ in-degree are almost half of the nodes having ‘0’ out-degree.

9. Calculate the clustering coefficient of each node

Formula used is :

For directed graphs, the clustering is similarly defined as the fraction of all possible directed triangles or geometric average of the subgraph edge weights for unweighted and weighted directed graph respectively [3].

$$c_u = \frac{1}{deg^{tot}(u)(deg^{tot}(u) - 1) - 2deg^{\leftrightarrow}(u)} T(u),$$

where $T(u)$ is the number of directed triangles through node u, $deg^{tot}(u)$ is the sum of in degree and out degree of u and $deg^{\leftrightarrow}(u)$ is the reciprocal degree of u.

Source :
https://core.ac.uk/download/pdf/54929154.pdf
https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.cluster.clustering.html

Clustering Coefficient

```
[28]  1 A = adjacency_matrix
      2 A_t = A.transpose()
      3
      4 temp = A + A_t
      5 temp1 = np.dot(temp,temp)
      6 d_num = np.dot(temp1,temp)
      7
      8 d_reciprocal = np.dot(A,A)
```

```
[34]  1 def cluster_coeff(A):
      2   clustering_coeff = defaultdict(lambda:0.0)
      3
      4   n = len(A)
      5   for i in range(n):
      6     degree_i = out_count[i] + in_count[i]
      7
      8     if(degree_i < 2):
      9       clustering_coeff[i] = 0              # node with 0 neighbours has clustering coeff = 0
     10     else:
     11       clustering_coeff[i] = d_num[i][i]/(2*degree_i*(degree_i - 1) - 2*d_reciprocal[i][i])
     12
     13   return clustering_coeff
```

```
[35]  1 clustering_coeff = cluster_coeff(adjacency_matrix)
```

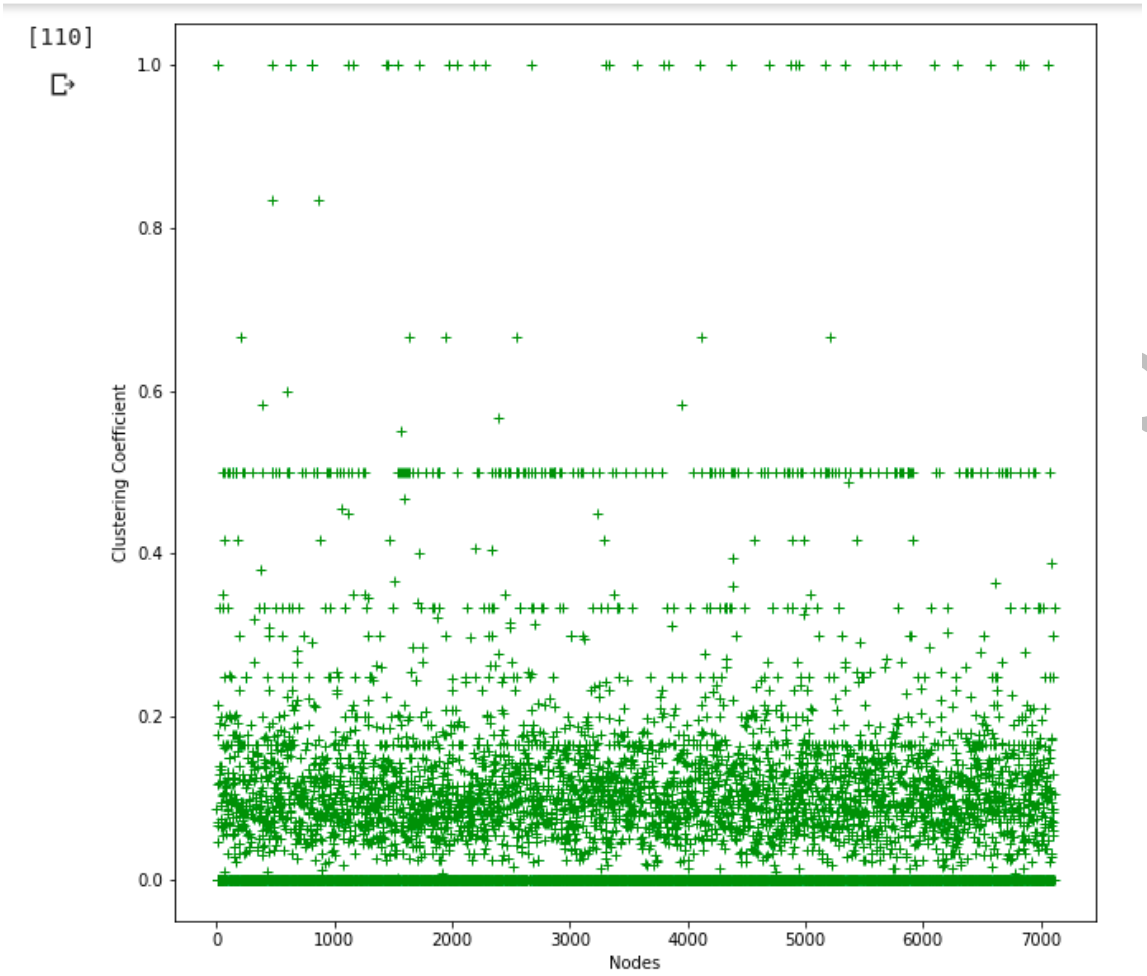
```
[36]  1 clustering_coeff
```

```
defaultdict(<function __main__.cluster_coeff.<locals>.<lambda>>,
            {0: 0.08797814207650273,
             1: 0.07033315705975675,
             2: 0.11659848042080655,
             3: 0.06666666666666667,
             4: 0.047619047619047616,
             5: 0,
             6: 0,
             7: 0,
             8: 0,
             9: 1.0,
             10: 0.17857142857142858,
             11: 0.21428571428571427,
             12: 0,
             13: 0.0,
             14: 0,
             15: 0,
             16: 0.0,
             17: 0,
             18: 0.12878787878787878,
             19: 0.3333333333333333,
             20: 0,
             21: 0,
             22: 0,
             23: 0,
             24: 0,
             25: 0,
             26: 0.13941102756892232,
             27: 0.19230769230769232,
             28: 0.11904761904761904,
             29: 0,
             30: 0,
             31: 0.2
```

Distribution of clustering coefficient

From the distribution of clustering coefficient given below, we can see that the number of nodes with clustering coefficient = 0 is very high and the number of nodes with clustering coefficient = 1 is fairly low. Majority of the nodes have a clustering coefficient between 0 and 0.2.

If every neighbour connected to node u is also connected to every other vertex within the neighbourhood of u, then this coefficient is 1. And it is 0 if no vertex that is connected to node u is connected to any other vertex that is connected to node u.



3. Degree centrality measure for each node

The degree centrality for a node u is the fraction of nodes it is connected to.

Source:
https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.degree_centrality.html#networkx.algorithms.centrality.degree_centrality

Formula used : total degree of the node/ total number of nodes
Total degree of a node = in-degree + out-degree

Degree Centrality

> The degree centrality for a node v is the fraction of nodes it is connected to.

```
[41]  1 deg_central = defaultdict(lambda:0.0)
      2
      3 for i in range(len(nodes)):
      4   k = out_count[i] + in_count[i]          # number of neighbouring nodes
      5   deg_central[i] = k/len(nodes)
```

```
[42]  1 deg_central
```

```
            526: 0.000140548137737175,
            527: 0.000140548137737175,
            528: 0.003935347856640899,
            529: 0.014195361911454674,
            530: 0.000140548137737175,
            531: 0.000140548137737175,
            532: 0.000140548137737175,
            533: 0.003935347856640899,
            534: 0.000140548137737175,
            535: 0.010119465917076598,
            536: 0.005200281096275475,
            537: 0.023330990864371046,
            538: 0.019536191145467324,
            539: 0.000140548137737175,
            540: 0.0037947997189037245,
            541: 0.012227687983134224,
            542: 0.0005621925509487,
            543: 0.000140548137737175,
            544: 0.000140548137737175,
            545: 0.00028109627547435,
            546: 0.002670414617006325,
            547: 0.001546029515108925,
```

Now, for the given dataset, we will calculate pagerank, authority and hub score of each node using the python library ‘networkx’.

For this, firstly a directed graph is generated from the adjacency matrix.

creating networkx graph from adjacency matrix

```
[ ]  1 G = nx.from_numpy_matrix(adjacency_matrix, parallel_edges = False, create_using=nx.DiGraph)
```

1. PageRank score for each node

```
[111]  1 rank = nx.pagerank(G)
```

```
[112]  1 rank
```

```
943: 5.048782345863015e-05,
944: 0.00023528233103830846,
945: 5.048782345863015e-05,
946: 5.048782345863015e-05,
947: 5.048782345863015e-05,
948: 5.048782345863015e-05,
949: 5.048782345863015e-05,
950: 0.0002927748895751096,
951: 5.048782345863015e-05,
952: 5.048782345863015e-05,
953: 5.048782345863015e-05,
954: 0.00010309515792015407,
955: 5.048782345863015e-05,
956: 5.048782345863015e-05,
957: 5.048782345863015e-05,
958: 0.0006076925581037549,
959: 0.0004756126138590225,
960: 5.048782345863015e-05,
961: 5.048782345863015e-05,
962: 5.048782345863015e-05,
963: 5.048782345863015e-05,
964: 5.048782345863015e-05,
965: 5.048782345863015e-05,
966: 0.00013945964879425653,
```

2. Authority and Hub score for each node

Hub and authority of each node

```
[28]  1 hub,authority = nx.hits(G)
```

```
      1 hub
```

```
455: 0.0001606043317284277 2,
456: 1.0543646435912817e-05,
457: 0.00013385403003626404,
458: 5.077552321526254e-07,
459: 7.014649976100327e-05,
460: 1.5361715105224924e-05,
461: 7.415283374988286e-06,
462: 7.415283374988286e-06,
463: 5.784324770865905e-05,
464: 7.415283374988286e-06,
465: 1.1020586623108884e-05,
466: 4.809069213627098e-06,
467: 0.0006643739957532808,
468: 1.537753905478071e-06,
469: 1.537753905478071e-06,
470: 1.537753905478071e-06,
471: 2.987418676609134e-05,
472: 1.537753905478071e-06,
473: 1.537753905478071e-06,
474: 2.1031060909498853e-06,
475: 1.537753905478071e-06,
476: 1.537753905478071e-06,
477: 0.0002417722135749167,
478: 0.0001345995065595588,
479: 0.00021244104862618157,
480: 0.00012608196354517917,
481: 0.00011421003713031443,
482: 3.690314353508838e-05,
483: 0.00022819903149893177,
484: 2.8877199802439765e-05,
```

```
[30]  1 authority
```

```
459: 7.014649976312264e-05,
460: 1.5361715105776194e-05,
461: 7.415283375204891e-06,
462: 7.415283375204891e-06,
463: 5.78432477095589e-05,
464: 7.415283375204891e-06,
465: 1.102058662344768e-05,
466: 4.809069213733387e-06,
467: 0.0006643739957624567,
468: 1.5377539055333056e-06,
469: 1.5377539055333056e-06,
470: 1.5377539055333056e-06,
471: 2.9874186766558777e-05,
472: 1.5377539055333056e-06,
473: 1.5377539055333056e-06,
474: 2.1031060909898703e-06,
475: 1.5377539055333056e-06,
476: 1.5377539055333056e-06,
477: 0.00024177221357774372,
478: 0.00013459950656094502,
479: 0.00021244104863095628,
480: 0.00012608196354836775,
481: 0.00011421003713220713,
482: 3.6903143535270674e-05,
483: 0.00022819903150288713,
484: 2.8877199803153675e-05,
485: 0.00029054458047670 77,
486: 0.00086838098062233 35,
487: 0.00049327528032077 84,
488: 2.0659004831060837e-05,
489: 0.00086801244724304 81,
490: 8.440333282858856e-06,
491: 0.00021770088610820055,
```

Comparison between the three obtained values:

PageRank

It is a method for rating Web pages and thus measures the human interest and attention devoted to those web pages. Ranking for every web page is computed based on the graph of the web.

Hits

This identifies good authorities and hubs for a topic by assigning two numbers to a page: an authority and a hub weight.

A higher authority weight occurs if the page is pointed to by pages with high hub weights.

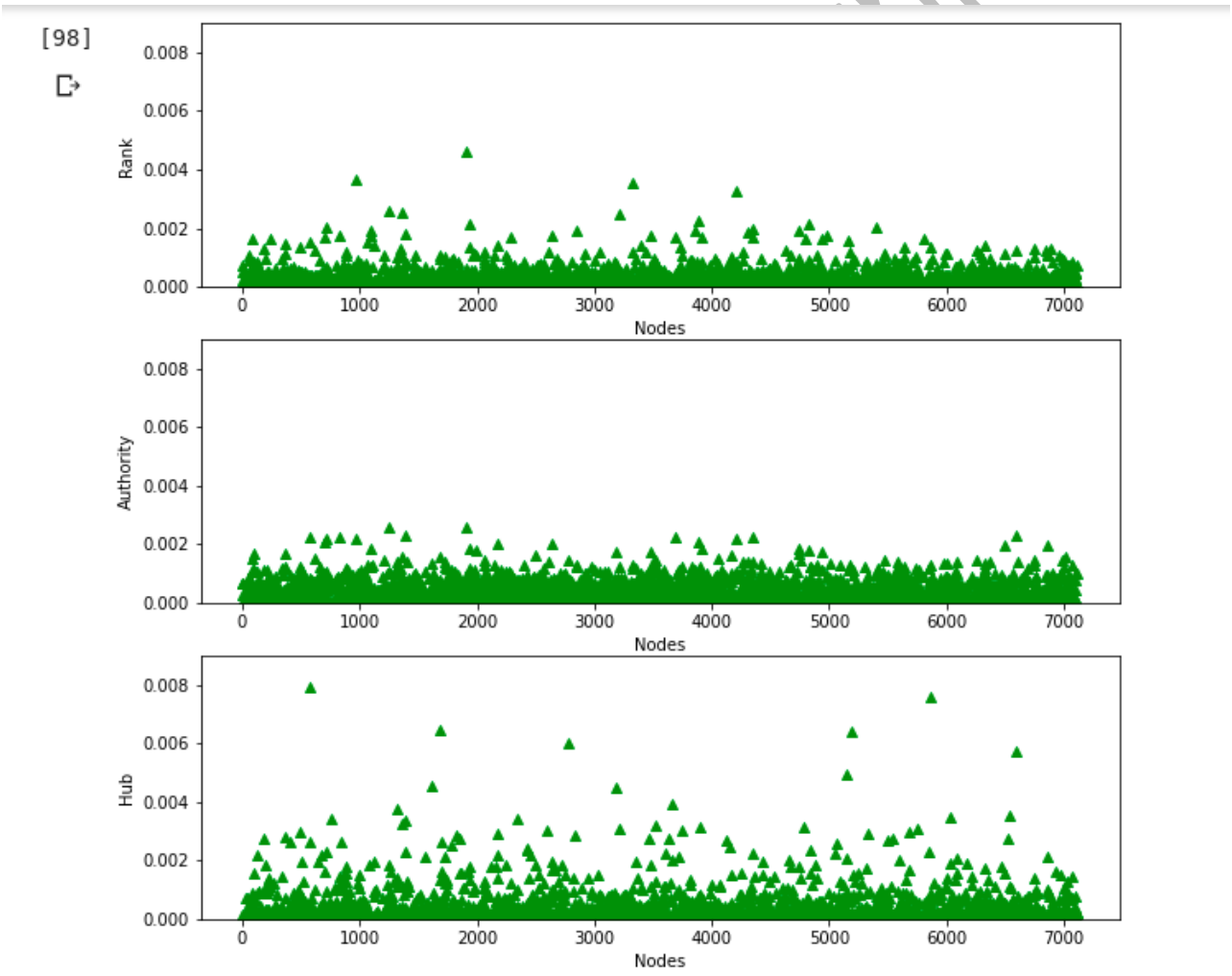
A higher hub weight occurs if the page points to many pages with high authority weights.

The advantage that pagerank has over Hits score is that query-time cost is lower in case of pagerank. It is less susceptible to localised link-spam.

The advantage that hit has over pagerank is that the ranking given by hits is query-sensitive. It includes the notion of hub and authority.

The difference is that unlike the PageRank algorithm, HITS only operates on a small subgraph (the seed S_Q) from the web graph. This subgraph is query dependent; whenever we search with a different query phrase, the seed changes as well.
HITS ranks the seed nodes according to their authority and hub weights.

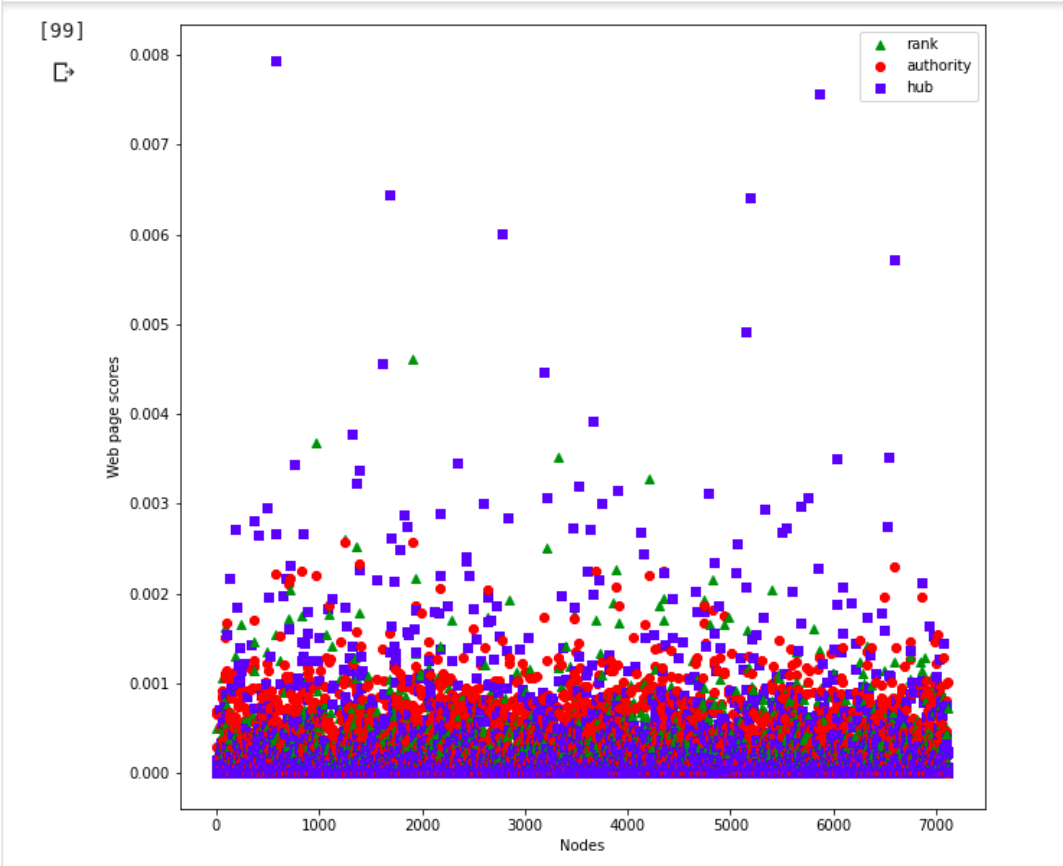
Following is the distribution of rank, authority and hub scores for each of the node:



The pagerank values lie between 0 and 0.005 only. Majority of the nodes have values accumulated between 0 and 0.001.

We can see that Authority values are accumulated towards the x-axis, i.e., most of them lie between 0 and 0.003.
While Hub values are very much scattered all over the space between 0 and 0.008.

Next is the visual comparison between all the three together:



Also, proper comparison between these scores can only be made if we are given a query to test. Because both of these use different strategies of assigning relevance to the web pages.

On fetching the top ten results obtained by pagerank, hub and authority, we get different results. This certifies the above fact.

finding top 10 results

```
[80]  1 top_rank = heapq.nlargest(10, rank.items(), key=lambda x: x[1])
      2 print(top_rank)
```

```
[(1902, 0.004612715891167531), (974, 0.00368122072952927), (3322, 0.0035248136576402542), (4206, 0.0032863743692308975), (1247, 0.0026053331
```

```
[81]  1 top_authority = heapq.nlargest(10, authority.items(), key=lambda x: x[1])
      2 print(top_authority)
```

```
[(1247, 0.0025801471780089l5), (1902, 0.0025732411241428006), (1393, 0.0023284150915378952), (6588, 0.0023037314804751022), (829, 0.0022558
```

```
[82]  1 top_hub = heapq.nlargest(10, hub.items(), key=lambda x: x[1])
      2 print(top_hub)
```

```
[(569, 0.007940492708074044), (5858, 0.00757433529744451), (1689, 0.006440248991012537), (5185, 0.006416870490195661), (2783, 0.00601056790
```

References:

http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture4/lecture4.html

https://nlp.stanford.edu/IR-book/html/htmledition/hubs-and-authorities-1.html